SPRING JAVA BASED CONFIGURATION

So far you have seen how we configure Spring beans using XML configuration file. If you are comfortable with XML configuration, then I will say it is really not required to learn how to proceed with Java based configuration because you are going to achieve the same result using either of the configurations available.

Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations explained below.

@Configuration & @Bean Annotations:

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context. The simplest possible @Configuration class would be as follows:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {

   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

Above code will be equivalent to the following XML configuration:

```
<beans>
   <bean   />
</beans>
```

Here the method name annotated with @Bean works as bean ID and it creates and returns actual bean. Your configuration class can have declaration for more than one @Bean. Once your configuration classes are defined, you can load & provide them to Spring container using *AnnotationConfigApplicationContext* as follows:

```
public static void main(String[] args) {
   ApplicationContext ctx =
   new AnnotationConfigApplicationContext(HelloWorldConfig.class);

   HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

   helloWorld.setMessage("Hello World!");
   helloWorld.getMessage();
}
```

You can load various configuration classes as follows:

```
public static void main(String[] args) {
   AnnotationConfigApplicationContext ctx =
   new AnnotationConfigApplicationContext();

   ctx.register(AppConfig.class, OtherConfig.class);
   ctx.register(AdditionalConfig.class);
   ctx.refresh();
```

```
   MyService myService = ctx.getBean(MyService.class);
   myService.doStuff();
}
```

Example:

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

Step	Description
1	Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project.
2	Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter.
3	Because you are using Java-based annotations, so you also need to add *CGLIB.jar* from your Java installation directory and *ASM.jar* library which can be downloaded from *asm.ow2.org*.
4	Create Java classes *HelloWorldConfig*, *HelloWorld* and *MainApp* under the *com.tutorialspoint* package.
5	The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Here is the content of **HelloWorldConfig.java** file:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {

   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

Here is the content of **HelloWorld.java** file:

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

Following is the content of the **MainApp.java** file:

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.annotation.*;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext ctx =
      new AnnotationConfigApplicationContext(HelloWorldConfig.class);

      HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

      helloWorld.setMessage("Hello World!");
      helloWorld.getMessage();
   }
}
```

Once you are done with creating all the source filesand adding required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, this will print the following message:

```
Your Message : Hello World!
```

Injecting Bean Dependencies:

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method calling another as follows:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class AppConfig {
   @Bean
   public Foo foo() {
      return new Foo(bar());
   }
   @Bean
   public Bar bar() {
      return new Bar();
   }
}
```

Here, the foo bean receives a reference to bar via constructor injection. Now let us see one working example:

Example:

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:

Step	Description
1	Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project.
2	Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter.
3	Because you are using Java-based annotations, so you also need to add *CGLIB.jar* from your Java installation directory and *ASM.jar* library which can be downloaded from *asm.ow2.org*.
4	Create Java classes *TextEditorConfig*, *TextEditor*, *SpellChecker* and *MainApp* under the *com.tutorialspoint* package.

5	The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Here is the content of **TextEditorConfig.java** file:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class TextEditorConfig {

   @Bean
   public TextEditor textEditor(){
      return new TextEditor( spellChecker() );
   }

   @Bean
   public SpellChecker spellChecker(){
      return new SpellChecker( );
   }
}
```

Here is the content of **TextEditor.java** file:

```
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker){
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }
   public void spellCheck(){
      spellChecker.checkSpelling();
   }
}
```

Following is the content of another dependent class file **SpellChecker.java**:

```
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling(){
      System.out.println("Inside checkSpelling." );
   }

}
```

Following is the content of the **MainApp.java** file:

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.*;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext ctx =
      new AnnotationConfigApplicationContext(TextEditorConfig.class);
```

```
      TextEditor te = ctx.getBean(TextEditor.class);

      te.spellCheck();
   }
}
```

Once you are done with creating all the source filesand adding required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, this will print the following message:

```
Inside SpellChecker constructor.
Inside TextEditor constructor.
Inside checkSpelling.
```

The @Import Annotation:

The **@Import** annotation allows for loading @Bean definitions from another configuration class. Consider a ConfigA class as follows:

```
@Configuration
public class ConfigA {
   @Bean
   public A a() {
      return new A();
   }
}
```

You can import above Bean declaration in another Bean Declaration as follows:

```
@Configuration
@Import(ConfigA.class)
public class ConfigB {
   @Bean
   public B a() {
      return new A();
   }
}
```

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied as follows:

```
public static void main(String[] args) {
   ApplicationContext ctx =
   new AnnotationConfigApplicationContext(ConfigB.class);
   // now both beans A and B will be available...
   A a = ctx.getBean(A.class);
   B b = ctx.getBean(B.class);
}
```

Lifecycle Callbacks:

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element:

```
public class Foo {
   public void init() {
      // initialization logic
   }
   public void cleanup() {
      // destruction logic
   }
}
```

```
@Configuration
public class AppConfig {
   @Bean(initMethod = "init", destroyMethod = "cleanup" )
   public Foo foo() {
      return new Foo();
   }
}
```

Specifying Bean Scope:

The default scope is singleton, but you can override this with the @Scope annotation as follows:

```
@Configuration
public class AppConfig {
   @Bean
   @Scope("prototype")
   public Foo foo() {
      return new Foo();
   }
}
```